Signals

Baris Simsek, 2005
http://www.enderunix.org/simsek/

ABOUT THIS DOCUMENT

You can find latest versions of this document at http://www.enderunix.org/simsek/

This document is an advanced introduction to UNIX traditional signals and POSIX.1 signals. Reliable and unreliable signal concepts will also be discussed throughout the document.

TRADITIONAL SIGNALS

Signals are software interrupts to notify a program. Applications generate signals to inform about their situation. Programmers can control asynchronous events by this way. For example, generally we cannot know when a child process terminates, it is truly random and can happen any time. When the child process exits, a signal will be generated to inform parent process.

Every signal has a name and a number. Following signals are the most popular signals programmers use:

```
 1 HUP  (hang up)
 2 INT  (interrupt)
 3 QUIT (quit)
 6 ABRT (abort)
 9 KILL (non-catchable, non-ignorable kill)
14 ALRM (alarm clock)
15 TERM (software termination signal)
```

UNIX Version 7 has 15 signals; SVR4 and 4.3BSD both have 31 different signals. You can find the complete list in manual pages. See signal(3).

Following conditions can generate a signal:

- When user presses terminal keys, the terminal will generate a signal. For example, when the user breaks a program by CTRL + C key pair.
- Hardware exceptions can generate signals, too: Division by 0, invalid memory reference. Inexperienced programmers often get SIGSEGV (Segmentation Violation signal) because of an invalid address in a pointer.
- Processes can send signals to themselves by using kill(2) system call (If permissions allow).
- Kernel can generate signal to inform processes when something happens. For example, SIGPIPE will be generated when a process writes to a pipe which has been closed by the reader.

Processes can send signal to other processes by using kill(2) system call. Every process can send signal in its privilege limitations. To send a signal, its real or efective user id has to be matched with

the receiver process. Superuser can send signals without any restriction.

```
NAME
     kill -- send signal to a process

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <sys/types.h>
     #include <signal.h>

     int
     kill(pid_t pid, int sig);
```

kill(1) is a simple command which allows us to send signals to other processes specified by a pid number. All UNIX like operating systems provide this command in their default installations.

```
NAME
     kill -- terminate or signal a process

SYNOPSIS
     kill [-s signal_name] pid ...
     kill -l [exit_status]
     kill -signal_name pid ...
     kill -signal_number pid ...
```

```
# ps x | grep sshd
  PID  TT  STAT      TIME COMMAND
  150  ??  Is     0:04.13 /usr/sbin/sshd

# kill -9 150
```

When a signal occurs, OS kernel has three alternative behaviours. These alternatives are named as dispositions or actions.

1. Kernel may ignore the signal. Notice that, SIGKILL and SIGSTOP cannot be ignored. Because, the superuser should have control over system processes. If a user process can ignore these two signals, superuser cannot kill or stop this process.
2. Catch the signal and wake a function to handle signal.
3. Apply the default action. The default action is mostly the terminatation of the process.

To handle a signal (2nd way) we must specify a handler function via signal(3).

```
NAME
     signal -- simplified software signal facilities

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <signal.h>

     void (*
     signal(int sig, void (*func)(int)))(int);
```

*func is a void function which takes signal number as a parameter.

Following code is from my sheffd project (http://cvsweb.enderunix.org/)

```
static void signal_handler(int signo)
{
      int status;

      switch(signo) {
            case SIGHUP:
                  syslog(LOG_INFO, "Restarting sheff daemon...");
                  loadconfig(cfgfile);
                  init_sheff();
                  break;
            case SIGCHLD:
                  /* wait for error */
                  if ((wait(&status)) == -1) return;
                  /* child has received signal */
                  if (WIFSIGNALED(status)) return;
                  /* child is stopped  */
                  if (WIFSTOPPED(status)) return;
                  /* successfull return */
                  if (WIFEXITED(status)) return;
                  break;
            case SIGTERM:
                  clean_exit();
                  /* blah blah, fill here */
                  break;
      }
}

int main()
{
      ...

      signal(SIGTTOU, SIG_IGN);
      signal(SIGTTIN, SIG_IGN);
      signal(SIGHUP, signal_handler);
      signal(SIGPIPE, signal_handler);
      signal(SIGTERM, signal_handler);
      signal(SIGCHLD, signal_handler);

      ...

      return 0;
}
```

You can also use SIG_IGN or SIG_DFL as a signal handler. These are defined in the signal.h as contants:

```
#define SIG_DFL ((__sighandler_t *)0)
#define SIG_IGN ((__sighandler_t *)1)
```

Use SIG_DFL to enable default handler for this signal; Use SIG_IGN to ignore this signal.

To handle SIGKILL, SIGHUP etc. we are using signal_handler() function. This function takes signal number as its parameter. You can specify what you want with this signal in the function.

POSIX.1 SIGNALS

Up until here, we've discussed traditional UNIX signals: Version 7 UNIX with traditional UNIX. 4.3BSD and SVR3 made some improvements to the signal model and introduced “reliable signals”. But that gave birth to incompatibilities between standards. POSIX.1 (from IEEE standards family) standardizes reliable signals based on 4.3BSD, as reliable signals.

What is the meaning of “reliable” or “unreliable”?

Programmers don't have complete control over unreliable signals: they can get lost. For example, they can be set to be “catched” or “ignored”, however, there are no means to “block” them. Following code snippet is a traditional example to point to problems with unreliable signals.

```
int flag = 0;

main()
{
  ...
  signal(SIGINT, handler);
  ...
  while(flag == 0) pause();
  ...
}

handler()
{
  signal(SIGINT, handler);
  flag = 1;
}
```

We want to ignore the signal but also want to know if it occurred. We are using a flag to keep this information. When a signal occurs, signal handler will set the flag to any non-zero value. pause() allows the process to sleep until a signal is received. After signal, process will continue.

If a signal occurs after test (flag == 0), we have some trouble. After the test, code reaches pause() call. Remember that the flag is still zero. At this point a signal occurs. So the flag is 1 now. After the signal, code will continue execution from pause() call. The flag is 1 right now. Program goes to sleep, because of pause. If the process doesn't receive any signals after that, it can go to sleep forever.

There are numerous examples about this.

RELIABLE SIGNALS

At first, it will be better if we define the some of the important concepts:

- When a signal occurs, we say signal is *generated*.
- We define an *action* for required signals.
- When an action is taken for a signal, this means signal is *delivered*.
- If a signal is between generation and delivery, this means signal is *pending*.
- We can *block* a signal for a process. If the process doesn't ignore the blocked signal, the signal will be pending.
- A blocked signal can be generated more than once before the process unblocks the signal. The kernel can deliver the signal once or more. If it delivers signals more than once, we say the signal

is *queued*. If the signals are delivered only once, it is not queued.

- Each process has a signal mask. Signal masks define blocked signals for a process. It is just a bit array which includes one bit for each signal. If the bit is on, that means related signal will be blocked.
- POSIX defines a data structure which holds one bit for each possible signal. This data structure is named *signal set*. It is an integer like data type. An integer is at least 32 bit value in 32 bit architectures. So we can hold status information for 32 different signals. We can define signal mask as a signal set. We can manage them by using some custom functions. POSIX defines some functions to modify any given signal set.

SIGNAL SETS

Signal sets are being used to represent multiple signals in a data structure.

```
sigset_t *set
```

It is not a feasible solution to allocate an integer for each signal. In some architectures, number of signals can exceed the number of bits in an integer. Signal sets solve this portability problem.

POSIX has these functions to manipulate signal sets:

```
NAME
     sigemptyset, sigfillset, sigaddset, sigdelset, sigismember --
     manipulate signal sets

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <signal.h>

     int
     sigemptyset(sigset_t *set);

     int
     sigfillset(sigset_t *set);

     int
     sigaddset(sigset_t *set, int signo);

     int
     sigdelset(sigset_t *set, int signo);

     int
     sigismember(const sigset_t *set, int signo);
```

sigemptyset() function initializes a signal set to be empty. This will cause exclusion of all signals.

sigfillset() function initializes a signal set to contain all signals. This will cause inclusion of all signals.

sigaddset() function adds the specified signal signo to the signal set.

sigdelset() function deletes the specified signal signo from the signal set.

sigismember() function returns whether a specified signal signo is contained in the signal set or not.

sigprocmask()

```
NAME
     sigprocmask -- manipulate current signal mask

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <signal.h>

     int
     sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

This function manipulates current signal mask which shows currently blocked signals. If the *set* is specified, the function installs new signal set instead of the current one. '*how*' indicates how the current mask is modified. It can be SIG_BLOCK, SIG_UNBLOCK or SIG_SETMASK. If SIG_BLOCK is selected, the new mask is the union of the current mask and the specified set. If SIG_UNBLOCK is selected, the new mask is the intersection of the current mask and the complement of the specified set. If *how* is SIG_SETMASK, the current mask is replaced by the specified set.

If *set* is NULL, *how* won't be affected and the process signal mask will not be changed.

If *oset* is not NULL, it preserves the previous value of the signal mask.

Notice that, SIGKILL and SIGSTOP cannot be blocked as we've discussed before.

sigpending()

```
NAME
     sigpending -- get pending signals

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <signal.h>

     int
     sigpending(sigset_t *set);
```

It returns pending signals as a signal set. Remember: if a signal is between generation and delivery, signal is called *pending*. '*set'* parameter will include returned set.

sigaction()

A process may specify a handler for a particular signal by using sigaction. It is similar to the signal() call discussed in unreliable signals.

This function supersedes the signal function from earlier releases of UNIX. In modern *BSD systems, the signal() facility is a simplified interface to the more general

```
NAME
     sigaction -- software signal facilities

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <signal.h>

     struct  sigaction {
             void (*sa_handler)();       /* signal handler */
             int sa_flags;               /* see signal options below */
             sigset_t sa_mask;           /* signal mask to apply */
     };

     int
     sigaction(int sig, const struct sigaction * restrict act,
         struct sigaction * restrict oact);
```

sig, specifies the signal number.

The *act* is a structure which is indicating the signal handler function and some other options. If the *act* is not NULL, the action will be modified. *'oact'* may be specified to save the previous action.

The *act* structure is defined like above. Linux has some options that are incompatible with POSIX.

Signal handler can be SIG_IGN or SIG_DFL instead of a function.

sa_flags item provides various options to handle signals. See manual page of sigaction(2) to get a complete list of flags. I'd like to talk about SA_RESTART which seems to be the most interesting of all.

Some system calls are slower than others. System calls are categorized into two sections: *slow* and others. ioctl, read, readv, write, writev, wait and waitpid can be given examples to slow system calls.

If a process is in a slow system call, it is possible that the process may be blocked for a long time or forever. For example, an interactive console program reads input from terminal and process the input. If the user walks away from terminal, read() system call will be blocked until he comes back and types something. If you get a signal delivered to your process, you can interrupt that slow system call. Before calling read(), you may call alarm(2) to set up a timer, and when the alarm goes off, SIGALRM signal will be delivered to the process, therefore, system call will return with the global error number (*errno)* set to EINTR.

The problem with slow system calls is that, we have to check the return value explicitly.

```
loop:
     if(( bcount = read(fd, buffer, sizeof(buffer)-1)) < 0) {
          /* If read is interrupted */
          if(errno == EINTR) goto loop;
          /* if read is not interrupted, it means an error occurred */
          printf("read error: %s\n", strerror(errno));
     }
```

If a signal is caught during the slow system call, the call may be forced to terminate with the error EINTR, the call may return with a data transfer shorter than requested, or the call may be restarted.

The SA_RESTART option comes with 4.2BSD and allows us to restart slow system calls. So we don't need to check return value.

Following simple code is an implementation of POSIX.1 interface:

```
#include <signal.h>

void handler()
{
        printf("in signal handler.n");[1]

        return;
}

int main()
{
        sigset_t newmask, oldmask;
        struct sigaction act, oact;

        act.sa_handler = handler;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;

        if(sigaction(SIGUSR1, &act, &oact) < 0)
                exit(-1);
        while(1) {
                continue;
        }

        return 0;
}
```

REFERENCES

- Advanced Programming in the UNIX Environment by W. Richard Stevens, Addison-Wesley

- FreeBSD 5.3 Library Functions Manuals

TODO

- Signal handling with threads.

- Kernel implementation of signals

Oct 16, 2005

Baris Simsek
simsek ~ enderunix.org
http://www.enderunix.org/simsek/
EnderUNIX Software Development Team

1 Using printf() here is only for demonstration purposes. Don't use it in a real programs signal handler. Please see the Single UNIX Specification for the list of allowed functions (http://www.opengroup.org/onlinepubs/009695399/).